Numele si prenumele (cu MAJUSCULE): _____ Grupa: _____

Test L3: _____ Tema: _____ Colocviu: _____ FINAL: _____

Test de laborator - Arhitectura Sistemelor de Calcul

17 ianuarie 2023

Varianta 1

- Nota maxima pe care o puteti obtine este 10.
- Nota obtinuta trebuie sa fie minim 5 pentru a promova, fara nicio rotunjire superioara.
- Orice tentativa de frauda este considerata o incalcare a Regulamentului de Etica!

1 Partea 0x00 - maxim 4p

Consideram ca vrem sa aplicam un algoritm de *clustering* intr-un graf ponderat (in care muchiile au costuri asociate). Algoritmul va grupa intr-un *cluster* (componenta conexa) acele varfuri care sunt cele mai apropiate din punctul de vedere al costului muchiilor dintre ele, fara sa se depaseasca o anumita valoare prestabilita. De exemplu, daca distanta maxima pentru a putea considera doua varfuri intr-o componenta conexa este 10, si intre varfurile `0` si `2` avem o muchie de cost 15, nu putem pune varfurile `0` si `2` in aceeasi componenta. Pentru a rezolva aceasta problema, se considera o varianta modificata a algoritmului Kruskal, care obtine arborele partial de cost minim. In contextul dat, impunand conditiile de mai sus, vom obtine mai multi arbori partiali de cost minim.

Consideram ca avem implementata o procedura `(long*, long) Kruskal(long *matrix)` care primeste ca intrare o matrice de costuri si returneaza: 1. un *array* de elemente de tip `.long` care, pe fiecare pozitie `i` contine *clusterul* in care se afla nodul cu indexul `i`; si 2. dimensiunea acestui *array*. De exemplu, putem obtine ca retur din procedura `([0 1 0 2 0 3 1 2], 8)`, ceea ce inseamna ca nodul 0 a ajuns in componenta conexa 0, nodul 1 in componenta conexa 1, nodul 2 in componenta conexa 0 etc.

Subiectul 1 (3p) Implementati o procedura `long countClusters(long *matrix)` care primeste ca argument matricea de costuri si returneaza numarul de clustere obtinute in urma aplicarii algoritmului Kruskal modificat. In `countClusters` vom avea un apel intern cate procedura `Kruskal`. Se garanteaza ca vom indexa componentele conexe in ordine, incepand cu 0. (in exemplul anterior, sunt 4 *clustere* formate) **Incepeti sa scrieti rezolvarea de pe aceasta pagina, si apoi folositi caseta de pe pagina urmatoare, in cazul in care mai aveti nevoie de spatiu**

Solution:

```
countClusters:
    push %ebp
    movl %esp, %ebp

    movl -8(%ebp), %eax

    push %eax
    call Kruskal
    addl $4, %esp

    xorl %edx, %edx
    push %ebx
    push %esi
    movl (%eax, %edx, 4), %esi
et_for:
    cmpl %edx, %ecx
    je et_exit

    movl (%eax, %edx, 4), %ebx
    cmpl %esi, %ebx
    jg change_max

et_cont:
    incl %edx
    jmp et_for

change_max:
    movl %ebx, %esi
    jmp et_cont

et_exit:
    movl %esi, %eax
    incl %eax

    pop %esi
    pop %ebx
    pop %ebp
    ret
```

Subiectul 2 (1p) Sa se reprezinte configuratia stivei in momentul in care are adancimea maxima, pentru procedura `countClusters`. Stim ca procedura `Kruskal` respecta toate conventiile de apel si utilizeaza intern toti registrii, precum si 3 variabile locale.

Solution: Solutia pentru procedura propusa (depinde de implementare!):

```
%esp:
    (<variabila locala 3 Kruskal>)
    (<variabila locala 2 Kruskal>)
    (<variabila locala 1 Kruskal>)
    (%edi Kruskal)
    (%esi Kruskal)
    (%ebx Kruskal)
%ebpKruskal:
    (%ebp Kruskal)
    (<r.a. Kruskal>)
    (*matrix)
%ebpClusters:
    (%ebp countClusters)
    (<r.a.countClusters>)
    (*matrix)
```

2 Partea 0x01 - maxim 3.5p

Subiectul 1 (0.7p) Fie urmatoarea secventa de cod:

```
et0:
    movl $14, %eax
    shr $4, %eax
    decl %eax
    xorl %edx, %edx
    cmpl %edx, %eax
    jb et1
    jmp et2
et1:
    movl $1, %ebx
    jmp et3
et2:
    movl $2, %ebx
et3:
```

Executam, in *debugger*, secvential, urmatoarele comenzi:

a. `b et0; run; stepi; stepi; i r eax`

b. `b et0; b et3; run; stepi; c; i r ebx`

Scrieti valoarea afisata la finalul fiecarui set de comenzi.

Solution: a. in eax se gaseste valoarea 0, este shiftare la dreapta cu 4, deci impartire cu $2^4 = 16$; b. ni se cere valoarea lui %ebx la eticheta `et3`, asa ca vom continua rationamentul programului: dupa ce %eax devine 0, este decrementat, si este comparat cu %edx, care este egal cu 0. Comparatia utilizata este pe **unsigned**, astfel ca se compara `0xffffffff` si `0`. Evident, `0xffffffff` este mai mare, si se face astfel salt la et2, unde %ebx primeste valoarea 2.

Subiectul 2 (0.7p) Fie urmatorul sir de instructiuni:

```
mov $0x30000000, %eax
mov $0x20, %ebx
mul %ebx
mov $0x8, %ecx
div %ecx
```

Ce valori vom gasi in registrii `%eax`, respectiv `%edx` dupa ce executam cele 5 instructiuni? Ce s-ar intampla daca am inlocui instructiunea `mov $0x8, %ecx` cu `mov $0x2, %ecx`?

Solution: Executam codul pas cu pas. Valoarea `$0x30000000` este $3 \cdot 16^7 = 3 \cdot 2^{28}$, iar in urma executarii urmatoarei linii, vom gasi ca `%eax` $= 3 \cdot 2^{28} \cdot 2^5$ (`$0x20` $= 2 \cdot 16 = 2^5$), deci `%eax` $= 3 \cdot 2^{33} = 6 \cdot 2^{32}$, de unde aflam ca `%eax = 0` si `%edx = 6` pana in momentul efectuarii impartirii. Continuam cu impartirea, avem valoarea 8 pusa in %ecx (deci 2^3), iar `div %ecx` va produce `(%edx, %eax) = (%edx, %eax) / %ecx`, deci `(%edx, %eax)` $= (0, \frac{3 \cdot 2^{33}}{2^3}) = (0, 3 \cdot 2^{30})$ (0 pe prima pozitie pentru ca impartirea este exacta, restul obtinut este 0), ceea ce inseamna in reprezentare binara pentru cat `1100...00`, doi `1` urmat de 30 de `0`, adica `0xc0000000`. Obtinem ca `%edx` este 0, iar `%eax` este `0xc0000000`. Daca incercam sa impartim doar la 2, rezultatul este $3 \cdot 2^{32}$ (rest 0), dar catul nu este reprezentabil pe 32 de biti, astfel ca vom obtine o exceptie aritmetica.

Subiectul 3 (0.7p) Justificati aparitia erorii `Segmentation fault` in momentul in care efectuati un apel `printf`, dar omiteti utilizarea simbolului `$` pentru sirul de format.

Solution: procedura incearca sa citeasca continutul de memorie de la adresa indicata. Daca nu folosim simbolul $, va incerca sa citeasca un continut de memorie care se afla la o adresa la care programul nostru nu are drepturi.

Subiectul 4 (0.7p) Consideram ca avem functia `f` cu doua argumente de tip `.long`, care utilizeaza local registrii `%eax`, `%ebx`, `%ecx`, `%esi` si `%edi`, precum si 7 variabile locale de tip `.long`. Aceasta functie este **recursiva** si respecta, in implementare, toate conventiile pentru crearea cadrului de apel prezentate la laborator. Stiind ca spatiul stivei programului vostru variaza intre `0xffdf4000` (inferior) si `0xffdf2000` (superior), dupa cate **autoapeluri** se va depasi spatiul alocat stivei?

Solution: Se determina mai intai spatiul de care dispunem, si anume diferenta dintre `0xffdf4000` si `0xffdf2000`, care este `0x2000`, ceea ce in zecimal inseamna $2 \cdot 16^3 = 8192$. Aceasta valoare este exprimata in Bytes. Determinam acum cat ocupa, in Bytes, stiva locala pentru cadrul de apel. Pentru procedura noastra intr-o configuratie locala avem cele doua argumente + adresa de retur + salvarea ebp-ului + salvarea %ebx + salvarea %esi + salvarea %edi + 7 variabile locale, insemnand 14 spatii, adica 56B. Facem impartirea, iar 8192 / 56 = 146 rest 16, deci la al 147-lea autoapel vom depasi acest spatiu alocat.

Subiectul 5 (0.7p) Fie urmatorul cod scris in limbajul `C`. Descrieti care va fi efectul executarii codului din `main`, utilizand cunostintele de Assembly din acest semestru.

```
void f()
{
    long x = 5;
}
void g()
{
    long y;
    printf("%d", y);
}
```

```
int main()
{
    f();
    g();

    return 0;
}
```

Solution: Afiseaza 5, pentru ca atat x, cat si y, vor fi acelasi element in stiva, y-ul din g va citi ce exista deja la -4(%ebp), care a fost deja completat cu 5 de functia f.

3 Partea 0x02 - maxim 2.5p

Presupunem ca aveti acces la un executabil `exec`, pe care il inspectati cu `objdump -d exec`. In momentul in care rulati aceasta comanda, va opriti asupra urmatorului fragment de cod. Analizati acest cod si raspundeti la intrebarile de mai jos. Pentru fiecare raspuns in parte, veti preciza si liniile de cod / instructiunile care v-au ajutat in rezolvare.

```
<func>:
     1.    pushl   %ebp
     2.    movl    %esp, %ebp
     3.    subl    $20, %esp
     4.    movl    24(%ebp), %eax
     5.    movb    %al, -20(%ebp)
     6.    movb    $0, -1(%ebp)
     7.    movl    $0, -8(%ebp)
.L4:
     8.    movl    -8(%ebp), %eax
     9.    cmpl    12(%ebp), %eax
    10.    jge     .L2
    11.    movl    -8(%ebp), %eax
    12.    cmpl    16(%ebp), %eax
    13.    jle     .L3
    14.    movl    -8(%ebp), %eax
    15.    cmpl    20(%ebp), %eax
    16.    jge     .L3
    17.    movl    -8(%ebp), %edx
    18.    movl    8(%ebp), %eax
    19.    addl    %edx, %eax
    20.    movzbl  (%eax), %eax
    21.    cmpb    %al, -20(%ebp)
    22.    jle     .L3
    23.    movl    -8(%ebp), %edx
    24.    movl    8(%ebp), %eax
    25.    addl    %edx, %eax
    26.    movzbl  (%eax), %eax
    27.    movb    %al, -1(%ebp)
.L3:
    28.    addl    $1, -8(%ebp)
    29.    jmp     .L4
.L2:
    30.    movzbl  -1(%ebp), %eax
    31.    ret
```

a. (0.5p) Cate argumente primeste procedura de mai sus?

Solution: 5 argumente - observam la linia 4 un `24(%ebp)`, deci avem 8, 12, 16, 20, 24 - argumentele procedurii, ca offset relativ la ebp

b. (0.5p) Stiind ca `movzbl` efectueaza un `mov` cu o conversie de tip, de la `.byte` la `.long`, ce tip de date returneaza aceasta procedura?

Solution: trebuie sa urmarim registrul `%eax`: ultima lui aparitie este la linia 30, unde se face un mov din -1(%ebp). Cum -1(%ebp) reprezinta continutul de memorie de la adresa %ebp - 1, conchidem ca se returneaza un byte (un char).

c. (0.5p) In procedura sunt utilizate si variabile locale. Descrieti care este scopul variabilei locale situata la adresa `%ebp - 8`.

Solution: urmarim -8(%ebp) in codul primit. observam ca este initializat cu 0 la linia 7. Ulterior, la eticheta .L4, observam ca -8(%ebp) este pus in %eax, cu scopul de a fi comparat cu 12(%ebp). Pana aici conchidem ca -8(%ebp), si argumentul procedurii 12(%ebp) (cel de-al doilea argument al procedurii) sunt elemente de tip .long, si se compara cu cmpl. Relatia devine, in limbaj natural, daca -8(%ebp) este mai mare sau egal cu 12(%ebp), sare la .L2, unde se face

un return si-un exit. Pana in acest punct, observam ca -8(%ebp) este utilizat intr-o conditie importanta de finalizare a logicii procedurii. Continuam cu analiza, si observam o similaritate logica intre calupul de linii 11-13 si 14-16, si anume: in ambele cazuri, variabila locala -8(%ebp) este comparata cu alte doua argumente ale procedurii, si anume 16(%ebp), respectiv 20(%ebp). Daca este mai mic sau egal decat 16(%ebp), sau daca este mai mare sau egal decat 20(%ebp), sare la .L3 unde **este incrementat**, si se revine la .L4. Conchidem, deci, ca -8(%ebp) este un index intr-o structura repetitiva.

d. (0.5p) Care este tipul de date al primului argument al procedurii?

Solution: Primul argument se afla la 8(%ebp), deci trebuie sa urmarim aparitiile acestuia. Parcurgem liniar codul, si gasim 8(%ebp) la linia 18, mutat in registrul %eax. Observam ca %eax este modificat sa devina %eax + %edx, unde %edx este valoarea indexului curent dintr-o structura repetitiva, conform mov-ului de la linia 17 si conform a ceea ce am determinat anterior. Observam la linia 20 un detaliu important: se face un mov cu conversie, astfel incat %eax sa stocheze continutul de memorie de la adresa din %eax! Daca asamblam informatia, 8(%ebp) reprezinta o adresa de memorie pe care o putem modifica cu orice numar intreg (nu cu multipli de 2, 4 ...), si asupra careia aplicam un movzbl, ceea ce ne duce cu gandul la un char* (byte ptr).

e. (0.5p) Descrieti comportamentul procedurii de mai sus, utilizand ceea ce ati descoperit la subpunctele anterioare.

Solution: Urmarim rezultatele anterioare, si stim ca avem 5 argumente, primul un char*, apoi trei long-uri, la 12, 16, respectiv 20 relativ la ebp, si ramane sa descoperim cine era 24(%ebp). Il vedem la linia 4, este mutat in %eax, iar apoi doar %al este stocat intr-o variabila locala, situata la -20(%ebp), de unde putem stabili ca este un char. In plus, apare comparatia de la linia 21, unde ceea ce poate fi elementul curent din sir este comparat cu acest -20(%ebp), folosindu-se un cmpb. Avem o structura repetitiva, unde am stabilit urmatoarele:

```
f(arg1, arg2, arg3, arg4, arg5)
    index = 0
    while (index < arg2)
    {
        if (index > arg3 && index < arg4 && arg1[index] > arg5)
        {
            trebuie sa detectam ce face
        }
        else
        {
            index = index + 1
        }
    }
    return (trebuie sa aflam ce)
```

ne intereseaza ce se intampla daca nu se trece la pasul urmator din structura, si anume aem de analizat liniile 23-27. Observam ca punem indexul in %edx, primul argument - sirul de caractere - in %eax, adunam %edx la %eax, deci modificam adresa sa fie pe caracterul curent, punem caracterul curent in %eax (prin utilizarea continutului de memorie), iar apoi salvam caracterul (%al) la -1(%ebp), deci intr-o variabila locala de tip char (prima variabila locala in sensul de crestere al stivei). Daca ne uitam, tot acest -1(%ebp) este si cel returnat. Ceea ce inseamna ca daca se respecta conditia de mai sus, deci index ¿ arg3 si index ¡ arg4 si arg1[index] ¿ arg5, atunci exista o variabila locala care preia arg1[index], iar la final se returneaza respectiva variabila locala. Observam si ca, la linia 6, respectiva variabila locala este initializata cu 0, pentru cazul in care nu se respecta conditia in niciun punct din procedura.

```
f(arg1, arg2, arg3, arg4, arg5)
    index = 0
    result (de tip char) = 0
    while (index < arg2)
    {
        if (index > arg3 && index < arg4 && arg1[index] > arg5)
        {
            result = arg1[index]
        }
        index = index + 1
    }
    return result
```